Improved Implementation of Point Location in General Two-Dimensional Subdivisions*

Michael Hemmer, Michal Kleinbort, and Dan Halperin

Tel-Aviv University, Israel

Abstract. We present a major revamp of the point-location data structure for general two-dimensional subdivisions via randomized incremental construction, implemented in CGAL, the Computational Geometry Algorithms Library. We can now guarantee that the constructed directed acyclic graph $\mathcal{G}$ is of linear size and provides logarithmic query time. Via the construction of the Voronoi diagram for a given point set S of size n, this also enables nearest-neighbor queries in guaranteed $O(\log n)$ time. Another major innovation is the support of general unbounded subdivisions as well as subdivisions of two-dimensional parametric surfaces such as spheres, tori, cylinders. The implementation is exact, complete, and general, i.e., it can also handle non-linear subdivisions. Like the previous version, the data structure supports modifications of the subdivision, such as insertions and deletions of edges, after the initial preprocessing. A major challenge is to retain the expected $O(n \log n)$ preprocessing time while providing the above (deterministic) space and query-time guarantees. We describe an efficient preprocessing algorithm, which explicitly verifies the length $\mathcal{L}$ of the longest query path in $O(n \log n)$ time. However, instead of using $\mathcal{L}$, our implementation is based on the depth $\mathcal{D}$ of $\mathcal{G}$. Although we prove that the worst case ratio of $\mathcal{D}$ and $\mathcal{L}$ is $\Theta(n/\log n)$, we conjecture, based on our experimental results, that this solution achieves expected $O(n \log n)$ preprocessing time.

1 Introduction

Birn et al. [1] presented a structure for planar nearest-neighbor queries, based on Delaunay triangulations, named Full Delaunay Hierarchies (FDH). The FDH is a very simple, and thus light, data structure that is also very easy to construct. It outperforms many other methods in several scenarios, but it does not have a worst-case optimal behavior. However, it is claimed [1] that methods that do have this behavior are too cumbersome to implement and thus not available. We got challenged by this claim.

* This work has been supported in part by the 7th Framework Programme for Research of the European Commission, under FET-Open grant number 255827 (CGL—Computational Geometry Learning), by the Israel Science Foundation (grant no. 1102/11), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

In this article we present an improved version of CGAL's planar point location that implements the famous incremental construction (RIC) algorithm as introduced by Mulmuley [2] and Seidel [3]. The algorithm constructs a linear size data structure that guarantees a logarithmic query time. It enables nearest-neighbor queries in guaranteed $O(\log n)$ time via planar point location in the Voronoi Diagram of the input points. In Section 4 we compare our revised implementation for point location, applied to nearest neighbor search, against the FDH. Naturally, this is only a byproduct of our efforts as planar point location is a very fundamental problem in Computational Geometry. It has numerous applications in a variety of domains including computer graphics, motion planning, computer aided design (CAD) and geographic information systems (GIS).

Previous Work: Most solutions can only provide an expected query time of $O(\log n)$ but cannot guarantee it, in particular, those that only require $O(n)$ space. Some may be restricted to static scenes that do not change, while others can only support linear geometry.

Triangulation-based point location methods, such as the approaches by Kirkpatrick [4] and Devillers [5] combine a logarithmic hierarchy with some walk strategy. Both require only linear space and Kirkpatrick can even guarantee logarithmic query time. However, both are restricted to linear geometry, since they build on a triangulation of the actual input.

Many methods can be summarized under the model of the trapezoidal search graph as pointed out by Seidel and Adamy [6]. Conceptually, the initial subdivision is further subdivided into trapezoids by emitting vertical rays (in both directions) at every endpoint of the input, which is the fundamental search structure. In principal, all these solutions can be generalized to support input curves that are decomposable into a finite number of x-monotone pieces.

The *slabs method* of Dobkin and Lipton [7] is one of the earliest examples. Every endpoint induces a vertical wall giving rise to $2n+1$ vertical slabs. A point location is performed by a binary search to locate the correct slab and another search within the slab in $O(\log n)$ time. Preparata [8] introduced a method that avoids the decomposition into $n + 1$ slabs reducing the required space from $O(n^2)$ to $O(n \log n)$. Sarnak and Tarjan [9] went back to the slabs of Dobkin and Lipton and added the idea of persistent data structures, which reduced the space consumption to $O(n)$. Another example for this model is the separating chains method of Lee and Preparata [10]. Combining it with fractional cascading, Edelsbrunner et al. [11], achieved $O(\log n)$ query time as well. For other methods and variants the reader is referred to a comprehensive overview given in [12].

An asymptotically optimal solution is the randomized incremental construction (RIC), which was introduced by Mulmuley [2] and Seidel [3]. In the static setting, it achieves $O(n \log n)$ preprocessing time, $O(\log n)$ query time and $O(n)$ space, all in expectancy. As pointed out in [13], the latter two can even be worst-case guaranteed. It is also claimed there that one can achieve these worst-case bounds in an expected preprocessing time of $O(n \log^2 n)$, but no concrete proof is given. The approach is able to handle dynamic scenes; that is, it is possible to add or delete edges later on. This method is discussed in more detail in Section 2.

Contribution: We present here a major revision of the trapezoidal-map random incremental construction algorithm for planar point location in CGAL. As the previous implementation, it provides a linear size data structure for non-linear subdivisions that can handle static as well as dynamic scenes. The new version is now able to guarantee $O(\log n)$ query time and $O(n)$ space. Following recent changes in the "2D Arrangements" package [14], the implementation now also supports unbounded subdivisions as well as ones that are embedded on two-dimensional parametric surfaces. After a review of the RIC in Section 2, we discuss, in Section 3, the difference between the length $\mathcal{L}$ of the longest search path and the depth $\mathcal{D}$ of the DAG. We prove that the worst-case ratio of $\mathcal{D}$ and $\mathcal{L}$ is $\Theta(n/\log n)$. Moreover, we describe two algorithms for the preprocessing stage that achieve guaranteed $O(n)$ size and $O(\log n)$ query time. Both are based on a verification of $\mathcal{L}$ after the DAG has been constructed: An implemented one that runs in expected $O(n \log^2 n)$ time, and a more efficient one that runs in expected $O(n \log n)$ time. The latter is a very recent addition that was not included in the reviewed submission. However, the solution that is integrated into CGAL is based on a verification of $\mathcal{D}$. Based on our experimental results, we conjecture that it also achieves expected $O(n \log n)$ preprocessing time. Section 4 demonstrates the performance of the new implementation by comparing our point location in a Voronoi Diagram with the nearest neighbor implementation of the FDH and others. Section 5 presents more details on the new implementation. To the best of our knowledge, this is the only available implementation for guaranteed logarithmic query time point location in general two-dimensional subdivisions.

2 Review of the RIC for Point Location

We review here the random incremental construction (RIC) of an efficient point location structure, as introduced by [2,3] and described in [13,15]. For ease of reading we discuss the algorithm in case the input is in general position. Given an arrangement of n pairwise interior disjoint x-monotone curves, a random permutation of the curves is inserted incrementally, constructing the Trapezoidal Map, which is obtained by extending vertical walls from each endpoint upward and downward until an input curve is reached or the wall extends to infinity. During the incremental construction, an auxiliary search structure, a directed acyclic graph (DAG), is maintained. It has one root and many leaves, one for every trapezoid in the trapezoidal map. Every internal node is a binary decision node, representing either an endpoint p, deciding whether a query lies to the left or to the right of the vertical line through p, or a curve, deciding if a query is above or below it. When we reach a curve-node, we are guaranteed that the query point lies in the x-range of the curve. The trapezoids in the leaves are interconnected, such that each trapezoid knows its (at most) four neighboring trapezoids, two to the left and two to the right. In particular, there are no common x-coordinates for two distinct endpoints[1].

[1] In the general case all endpoints are lexicographically compared; first by the x-coordinate and then by the y-coordinate. This implies that two covertical points produce a virtual trapezoid, which has a zero width.

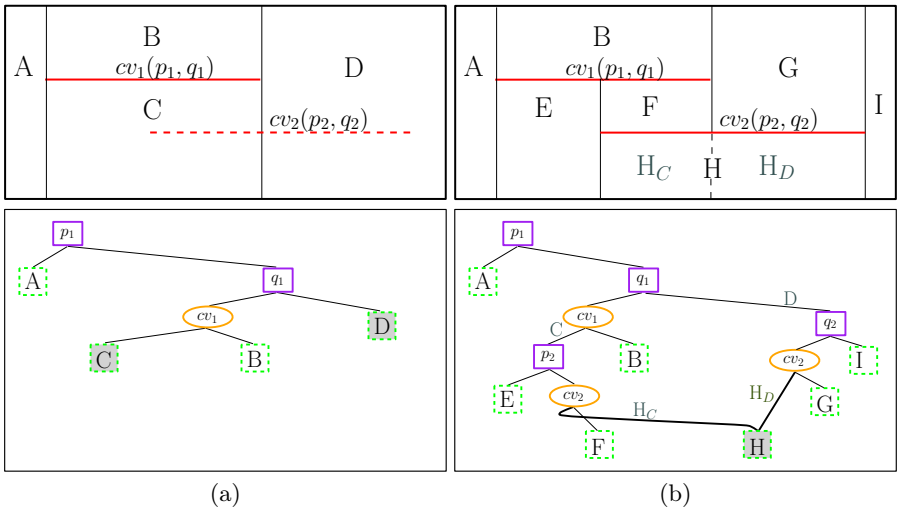


Fig. 1: Trapezoidal decomposition and the constructed DAG for two segments cv_1 and cv_2: (a) before and (b) after the insertion of cv_2. The insertion of cv_2 splits the trapezoids C, D into E, F, H_C and G, I, H_D, respectively. H_C and H_D are merged into H, as they share the same top (and bottom) curves.

When a new x-monotone curve is inserted, the trapezoid containing its left endpoint is located by a search from root to leaf. Then, using the connectivity mechanism described above, the trapezoids intersected by the curve are gradually revealed and updated. Merging new trapezoids, if needed, takes time that is linear in the number of intersected trapezoids. The merge makes the data structure become a DAG (as illustrated in Figure 1) with expected $O(n)$ size, instead of an $\Omega(n \log n)$ size binary tree [6]. For an unlucky insertion order the size of the resulting data structure may be quadratic, and the longest search path may be linear. However, due to the randomization one can expect $O(n)$ space, $O(\log n)$ query time, and $O(n \log n)$ preprocessing time.

3 On the Difference between Paths and Search Paths

As shown in [13], one can build a data structure, which guarantees $O(\log n)$ query time and $O(n)$ size, by monitoring the size and the length of the longest search path $\mathcal{L}$ during the construction. The idea is that as soon as one of the values becomes too large, the structure is rebuilt using a different random insertion order. It is shown that only a small constant number of rebuilds is expected. However, in order to retain the expected construction time of $O(n \log n)$, both values must be efficiently accessible. While this is trivial for the size, it is not clear how to achieve this for $\mathcal{L}$. Hence, we resort to the depth $\mathcal{D}$ of the DAG, which is an upper bound on $\mathcal{L}$ as the set of all possible search paths is a subset of all paths in the DAG. Thus, the resulting data structure still guarantees a logarithmic query time.

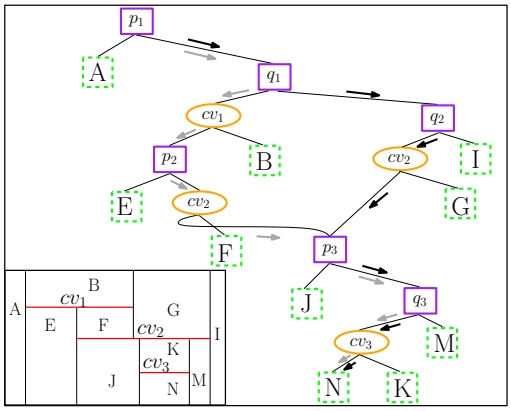
The depth $\mathcal{D}$ can be made accessible in constant time by storing the depth of each leaf in the leaf itself, and maintaining the maximum depth in a separate variable. The cost of maintaining the depth can be charged to new nodes, since

existing nodes never change their depth value. This is not possible for $\mathcal{L}$ while retaining linear space, since each leaf would have to store a non-constant number of values, i.e., one for each valid search path that reaches it. In fact the memory consumption would be equivalent to the data structure that one would obtain without merging trapezoids, namely the trapezoidal search tree, which for certain scenarios requires $\Omega(n \log n)$ memory as shown in [6]. In particular, it is necessary to merge as (also in practice) the sizes of the resulting search tree and the resulting DAG considerably differ.

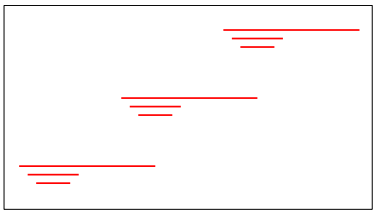
In Section 3.1 we show that for a given DAG its depth $\mathcal{D}$ can be linear while $\mathcal{L}$ is still logarithmic, that is, such a DAG would trigger an unnecessary rebuild. It is thus questionable whether one can still expect a constant number of rebuilds when relying on $\mathcal{D}$. Our experiments in Subsection 3.3 show that in practice the two values hardly differ, which indicates that it is sufficient to rely on $\mathcal{D}$. However, a theoretical proof to consolidate this is still missing. Subsection 3.2 provides efficient preprocessing solutions for the static scenario (where all segments are given in advance). As such, we see it as a concretization of, and an improvement over, the claim mentioned in [13].

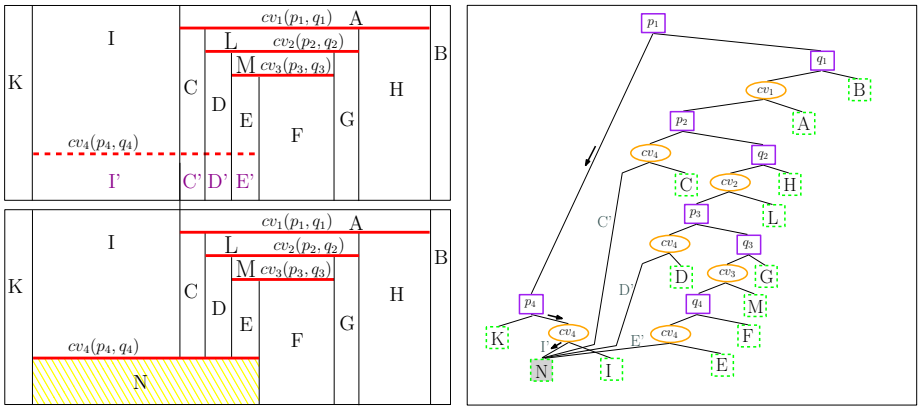
3.1 Worst Case Ratio of Depth and Longest Search Path

The figure to the right shows the DAG of Figure 1 after inserting a third segment. There are two paths that reach the trapezoid N (black and gray arrows). However, the gray path is not a valid search path, since all points in N are to the right of q_1; that is, such a search would never visit the left child of q_1. It does, however, determine the depth of N, since it is the longer path of the two. In the sequel we use this observation to construct an example that shows that the ratio between $\mathcal{D}$ and $\mathcal{L}$ can be as large as $\Omega(n/\log n)$. Moreover, we will show that this bound is tight.



We start by constructing a simple-to-demonstrate lower bound that achieves $\Omega(\sqrt{n})$ ratio between $\mathcal{D}$ and $\mathcal{L}$. Assuming that $n = k^2 \in \mathbb{N}$, the construction consists of k blocks, each containing k horizontal segments. The blocks are arranged as depicted in the figure to the right. Segments are inserted from top to bottom. A block starts with a large segment at the top, which we call the *cover segment*, while the other segments successively shrink in size. Now the next block is placed to the left and below the previous block. Only the cover segment of this block extends below the previous block, which causes a merge as illustrated in Figure 2. All $k = \sqrt{n}$ blocks are placed in this fashion. This

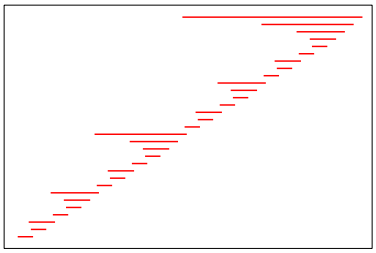




(a) (b)

Fig. 2: (a) The trapezoidal-map after inserting cv_4. The map is displayed before and after the merge of I', C', D', and E' into N, in the top and bottom illustrations, respectively. A query path to the region of I' in N will take 3 steps, while the depth of N in this example is 11.

construction ensures that each newly inserted segment intersects the trapezoid with the largest depth, which increases $\mathcal{D}$. The largest depth of $\Omega(n)$ is finally achieved in the trapezoid below the lowest segment. However, the actual search path into this trapezoid has only $O(\sqrt{n})$ length, since for each previous block it only passes through one node in order to skip it and $O(\sqrt{n})$ in the last block.



The following construction, which uses a recursive scheme, establishes the lower bound $\Omega(n/\log n)$ for $\mathcal{D}/\mathcal{L}$. Blocks are constructed and arranged in a similar fashion as in the previous construction. However, this time we have $\log n$ blocks, where block i contains $n/2^i$ segments. Within each block we then apply the same scheme recursively as depicted in the figure to the right. Again segments are inserted top to bottom such that the depth of $\Omega(n)$ is achieved in the trapezoid below the lowest segment. The fact that the lengths of all search paths are logarithmic can be proven by the following argument. By induction we assume that the longest path within a block of size $n/2^i$ is some constant times $(\log_2 n - i)$. Obviously this is true for a block containing only one segment. Now, in order to reach block i with $n/2^i$ segments, we require $i-1$ comparisons to skip the $i-1st$ preceding blocks. Thus in total the search path is of logarithmic length.

Theorem 1. *The $\Omega(n/\log n)$ worst-case lower bound on $\mathcal{D}/\mathcal{L}$ is tight.*

Proof. Obviously, $\mathcal{D}$ of $O(n)$ is the maximal achievable depth, since by construction each segment can only appear once along *any* path in the DAG. It remains to show that for any scenario with n segments there is no DAG for which $\mathcal{L}$ is

smaller than $\Omega(\log n)$. Since there are n segments, there are at least n different trapezoids having these segments as their top boundary. Let T be a decision tree of the optimal search structure. Each path in the decision tree corresponds to a valid search path in the DAG and vice versa. The depth of T must be larger than $\log_2 n$, since it is only a binary tree. We conclude that the worst case ratio of $\mathcal{D}$ and $\mathcal{L}$ is $\Theta(n/\log n)$. □

3.2 Efficient Solutions for Static Subdivisions

We first describe an algorithm for static scenes that runs in expected $O(n \log^2 n)$ time, constructing a DAG of linear size in which $\mathcal{L}$ is $O(\log n)$. The result is based on the following lemma.

Lemma 1. *Let S be a planar subdivision induced by n pairwise interior disjoint x-monotone curves. The expected size of the trapezoidal search tree $\mathcal{T}$, which is constructed as the RIC above but without merges, is $O(n \log n)$.*

Proof. Since $\mathcal{T}$ is binary tree, it is sufficient to bound the expected number of leaves in $\mathcal{T}$, namely, the number of trapezoids (without merges), which is bounded by twice the number of vertical edges + 1. First, focus on a vertical wall W induced by one endpoint of the ith inserted curve. W is intersected by n curves, in the worst-case. The $i-1$ already inserted curves partition W into i intervals. However, we are only interested in the interval I containing the endpoint of the ith curve, as it will appear in the final structure. Curves inserted after the ith curve may split I. The expected number of intersections in I (including the endpoint of the ith curve) is $O((n-i)/i)$. Summing up over all vertical walls gives a total of $O(n \log n)$ expected intersections. Thus, the expected number of vertical edges is $O(n \log n)$ as well, and , clearly, this is also the expected size of the tree. □

The following algorithm *compute_max_search_path_length* computes $\mathcal{L}$ in expected $O(n \log^2 n)$ time. Starting at the root it descends towards the leaves in a recursive fashion. Taking the history of the current path into account, each recursion call maintains the interval of x values that are still possible. Thus, if an x-coordinate of a point node is not contained in the interval the recursion does not need to split. This means that the algorithm essentially mimics $\mathcal{T}$ (as it would have been constructed), since the recursion only follows possible search paths. By Lemma 1 the expected number of leaves of $\mathcal{T}$, and thus of search paths, is $O(n \log n)$. Since the expected length of a query is $O(\log n)$ this algorithm takes expected $O(n \log^2 n)$ time.

Definition 1. *$f(n)$ denotes the time it takes to verify that, in a linear size DAG constructed over a planar subdivision of n x-monotone curves, $\mathcal{L}$ is bounded by $c \log n$ for a constant c.*

Theorem 2. *Let S be a planar subdivision with n x-monotone curves. A point location data structure for S, which has $O(n)$ size and $O(\log n)$ query time in the worst case, can be built in $O(n \log n + f(n))$ expected time, where $f(n)$ is as defined above.*

Proof. The construction of a DAG with some random insertion order takes expected $O(n \log n)$ time. The linear size can be verified trivially on the fly. After the construction the algorithm *compute_max_search_path_length* is used to verify that $\mathcal{L}$ is logarithmic. The verification of the size and $\mathcal{L}$ may trigger rebuilds with a new random insertion order. However, according to [13], one can expect only a constant number of rebuilds. Thus, the overall expected runtime remains expected $O(n \log n + f(n))$. □

The verification process described above takes expected $O(n \log^2 n)$ time. However, one can do better as we briefly sketch next. Let T be the collection of *all* the trapezoids created during the construction of the DAG, including intermediate trapezoids that are later killed by the insertion of later segments. Let $\mathcal{A}(T)$ denote the arrangement of the trapezoids. The *depth* of a point p in the arrangement is defined as the number of trapezoids in T that cover p. The key to the improved algorithm is the following observation by Har-Peled.

Observation 1. *The length of a path in the DAG for a query point p is at most three times the depth of p in $\mathcal{A}(T)$.*

We remark that this depth is established in an interior of a face of $\mathcal{A}(T)$ since we consider the boundaries of the trapezoids as open. This can be done since the longest path will always end in a leaf of the DAG, which represents a trapezoid. For any query point that falls on either a curve or an endpoint of the initial subdivision the search path will end in an internal node of the DAG. The search path for a query point q on a vertical edge of a trapezoid will be identical to a path for a query point in a neighboring trapezoid.

It follows that we need to verify that the maximum depth of a point in $\mathcal{A}(T)$ is some constant $c_1 \log n$. Since the input curves in S are interior pairwise disjoint, according to the separation property stemming from [16], one can define a total order on the curves. This order allows us to apply a modified version[2] of an algorithm by Alt and Scharf [17], which originally detects the maximum depth in an arrangement of n axis-parallel rectangles in $O(n \log n)$ time. Notice that we only apply this verification algorithm on DAGs of linear size. Putting everything together we obtain:

Theorem 3. *Let S be a planar subdivision with n x-monotone curves. A point location data structure for S, which has $O(n)$ size and $O(\log n)$ query time in the worst case, can be built in $O(n \log n)$ expected time.*

3.3 Experimental Results

Since $\mathcal{D}$ is an upper bound on $\mathcal{L}$ and since $\mathcal{D}$ is accessible in constant time our implementation explores an alternative that monitors $\mathcal{D}$ instead of $\mathcal{L}$. Though this may cause some additional rebuilds, the experiments in this section give strong evidence that one can still expect $O(n \log n)$ preprocessing time. We compared $\mathcal{D}$

[2] More details can be found in Appendix C.

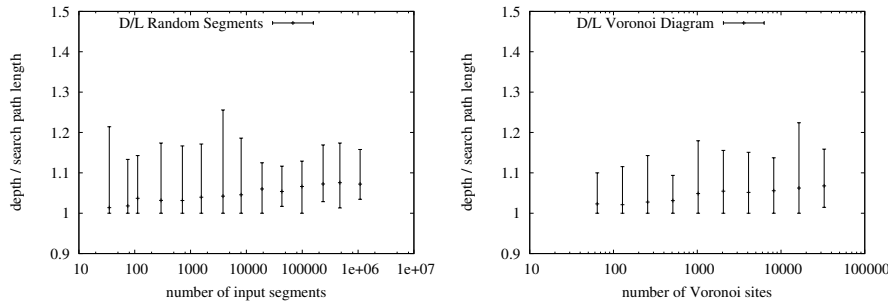


Fig. 3: Ratio of $\mathcal{D}$ and $\mathcal{L}$ in two scenarios: random segments (left), Voronoi diagram of random points (right). Plots show average value with error bars.

and $\mathcal{L}$ in two different scenarios: random non-intersecting line segments and Voronoi diagram for random sites.[3] Each scenario was tested with an increasing number of subdivision edges, with several runs for each input. Figure 3 displays the average $\mathcal{D}/\mathcal{L}$ ratio, and also the minimal and maximal ones. Obviously, the average ratio is close to 1 and never exceeded a value of 1.3.

These experimental results indicate that replacing the test for the length of the longest path $\mathcal{L}$ by the depth $\mathcal{D}$ of the DAG in the randomized incremental construction essentially does not harm the runtime. However, the following conjecture remains to be proven.

Conjecture 1. There exists a constant $c > 0$ such that the runtime of the randomized incremental algorithm, modified such that it rebuilds in case the depth $\mathcal{D}$ of the DAG becomes larger than $c \log n$, is expected $O(n \log n)$, i.e., the number of expected rebuilds is still constant.

4 Nearest Neighbor Search in Guaranteed $O(\log n)$ Time

As stated in the Introduction, we were challenged by the claim of Birn et al. [1] that guaranteed logarithmic nearest-neighbor search can be achieved via efficient point location on top of the Voronoi Diagram of the input points, but that this approach *"does not seem to be used in practice"*. With this section we would like to emphasize that such an approach is available and that it should be considered for use in practice. Using the RIC planar point location, the main advantage would be that query times are stable and independent of the actual scenario.

4.1 Nearest Neighbor Search via Voronoi Diagram

Given a set P of n points, which we wish to preprocess for efficient point location queries, we first create a Delaunay triangulation (DT) which takes $O(n \log n)$ expected time. The Voronoi diagram (VD) is then obtained by dualizing. Using a

[3] Appendix A contains additional experimental results that include also the scenarios constructed in Section 3.1.

sweep, the arrangement representing the VD, which has at most $3n - 6$ edges, can be constructed in $O(n \log n)$ time. However, taking advantage of the spatial coherence of the edges, we use a more efficient method that directly inserts VD edges while crawling over the DT. The resulting arrangement is then further processed by our RIC implementation. If Conjecture 1 is true then this takes expected $O(n \log n)$ time. Alternatively, it would have been possible to implement the solution presented in Subsection 3.2 (for which we can prove expected $O(n \log n)$ preprocessing time).

4.2 Nearest Neighbor Search via Full Delaunay Hierarchy

The full Delaunay hierarchy (FDH) presented in [1] is based on the fact that one can find the nearest neighbor by performing a greedy walk on the edges of the Delaunay triangulation (DT). The difference is that the FDH keeps all edges that appear during the randomized construction [18] of the DT in a flattened n-level hierarchy structure, where level i contains the DT of the first i points. Thus, a walk that starts at the first point is accelerated due to long edges that appeared at an early stage of the construction process while the DT was still sparse. The FDH is a very light, easy to implement, and fast data structure with expected $O(n \log n)$ construction time that achieves an expected $O(\log n)$ query path length. However, a query may take $O(n)$ time since the degree of nodes can be linear. For the experiments we used two exact variants: a basic exact version (EFDH) and a (usually faster) version (FFDH) that first performs a walk using inexact floating point arithmetic and then continues with an exact walk.

4.3 Experimental Results

We compared our implementation for nearest-neighbor search using the RIC point location on the Voronoi-diagram (ENNRIC) to the following exact methods: EFDH, FFDH, CGAL's Delaunay hierarchy (CGAL_DH) [5], and CGAL's kd-tree (CGAL_KD).[4]

All experiments have been executed on a Intel(R) Core(TM) i5 CPU M 450 with 2.40GHz, 512 kB cache and 4GB RAM memory, running Ubuntu 10.10. Programs were compiled using `g++` version 4.4.5 optimized with `-O3` and `-DNDEBUG`. The left plot of Figure 4 displays the total query time in a random scenario, in which both input points and query points are randomly chosen within the unit square. Clearly, all methods have logarithmic query time, however due to larger constants ENNRIC is slower. The other plot presents a combined scenario of $(n - \lfloor \log n \rfloor)$ equally spaced input points on the unit circle and $\lfloor \log n \rfloor$ random outliers. The queries are random points in the same region. In this experiment the CGAL_KD and ENNRIC are significantly faster and maintain a stable query time. A similar scenario that was tested contains equally spaced input points on a circle and a point in the center with random query points inside the circle. The differences there are even more significant than in the previous scenario. As for the preprocessing time in all tested scenarios, obviously ENNRIC cannot compete with the fast construction time of the other methods.

[4] Due to similar performance we elided the kd-tree implementation in ANN [19].

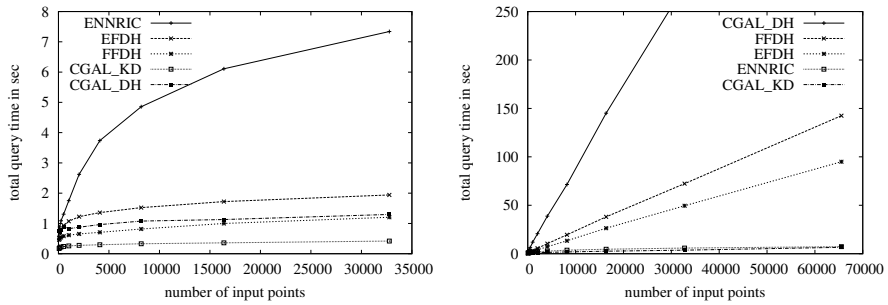


Fig. 4: Performance of 500k nearest-neighbor queries for different methods on two scenarios: (left) random points; (right) circle with outliers.

5 CGAL's New RIC Point Location

With this article we announce our revamp of CGAL's implementation of planar point location via the randomized incremental construction of the trapezoidal map, which is going to be available in the upcoming CGAL release 4.1.

Like the previous implementation by Oren Nechushtan [20], it is part of the "2D Arrangements" package [21] of CGAL. It allows both insertions and deletions of edges. The implementation is exact and covers all degenerate cases. Following the *generic-programming paradigm* [22] it can be easily applied to linear geometry but also to non-linear geometry such as algebraic curves or Bézier curves. The main new feature, and this is what triggered this major revision, is the support for unbounded curves, as it was introduced for the "2D Arrangements" package in [14], enabling point location on two-dimensional parametric surfaces (e.g., spheres, tori, etc.) as well.

In addition we did a major overhaul of the code basis. In particular, we maintain the depth $\mathcal{D}$ of the DAG as described in Section 3 such that $\mathcal{D}$ is accessible in constant time. Thus we can now guarantee logarithmic query time after every operation. Moreover, the data structure now operates directly on the entities of the arrangement. In particular, it avoids copying of geometric data which can significantly reduce the amount of additional memory that is used by the search structure. This is important, since due to the generic nature of the code it is not clear whether the geometric types (user provided) are referenced.

To the best of our knowledge, this is the only available implementation of a point location method with a guaranteed logarithmic query time that can handle two-dimensional subdivisions to this generality. Furthermore, it is the fastest available point location method, in terms of query time, for CGAL arrangements.[5]

6 Open Problem

Prove Conjecture 1, that is, prove that it is possible to rely on the depth $\mathcal{D}$ of the DAG and still expect only a constant number of rebuilds. This solution would not require any changes to the current implementation.

[5] A comparison to CGAL Landmarks point location [23] is given in the Appendix B.

Acknowledgement: The authors thank Sariel Har-Peled for sharing Observation 1, which is essential to the expected $O(n \log n)$ time algorithm for producing a worst-case linear-size and logarithmic-time point-location data structure.

References

1. Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and fast nearest neighbor search. In: Workshop on Algorithm Engineering and Experiments. (2010) 43–54
2. Mulmuley, K.: A fast planar partition algorithm, i. J. Symb. Comput. **10**(3/4) (1990) 253–280
3. Seidel, R.: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. J. Comput. Geom. **1** (1991) 51–64
4. Kirkpatrick, D.G.: Optimal search in planar subdivisions. SIAM J. Comput. **12**(1) (1983) 28–35
5. Devillers, O.: The Delaunay hierarchy. Int. J. Found. Comput. Sci. **13**(2) (2002) 163–180
6. Seidel, R., Adamy, U.: On the exact worst case query complexity of planar point location. J. Algorithms **37**(1) (2000) 189–217
7. Dobkin, D.P., Lipton, R.J.: Multidimensional searching problems. SIAM J. Comput. **5**(2) (1976) 181–186
8. Preparata, F.P.: A new approach to planar point location. SIAM J. Comput. **10**(3) (1981) 473–482
9. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. Commun. ACM **29**(7) (1986) 669–679
10. Lee, D.T., Preparata, F.P.: Location of a point in a planar subdivision and its applications. In: ACM Symposium on Theory of Computing (STOC). STOC '76, New York, NY, USA, ACM (1976) 231–235
11. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. SIAM J. Comput. **15**(2) (1986) 317–340
12. Snoeyink, J.: Point location. In Goodman, J.E., O'Rourke, J., eds.: Handbook of Discrete and Computational Geometry. CRC Press LLC, Boca Raton, FL (2004) 767–785
13. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Third edn. Springer-Verlag (2008)
14. Berberich, E., Fogel, E., Halperin, D., Melhorn, K., , Wein, R.: Arrangements on parametric surfaces I: General framework and infrastructure. Mathematics in Computer Science **4** (2010) 67–91
15. Mulmuley, K.: Computational geometry - an introduction through randomized algorithms. Prentice Hall (1994)
16. Guibas, L.J., Yao, F.F.: On translating a set of rectangles. In: STOC. (1980) 154–160
17. Alt, H., Scharf, L.: Computing the depth of an arrangement of axis-aligned rectangles in parallel. In: Proceedings of the 26th European Workshop on Computational Geometry (EuroCG), Dortmund, Germany (March 2010) 33–36
18. Amenta, N., Choi, S., Rote, G.: Incremental constructions con brio. In: Symposium on Computational Geometry. (2003) 211–219
19. Mount, D.M., Arya, S.: Ann: A library for approximate nearest neighbor searching `http://www.cs.umd.edu/~mount/ANN/`.

20. Flato, E., Halperin, D., Hanniel, I., Nechushtan, O., Ezra, E.: The design and implementation of planar maps in CGAL. ACM Journal of Experimental Algorithmics **5** (2000) 13
21. Wein, R., Berberich, E., Fogel, E., Halperin, D., Hemmer, M., Salzman, O., Zukerman, B.: 2D arrangements. In: CGAL User and Reference Manual. 4.0 edn. CGAL Editorial Board (2012)
22. Austern, M.H.: Generic Programming and the STL. Addison-Wesley (1999)
23. Haran, I., Halperin, D.: An experimental study of point location in planar arrangements in CGAL. ACM Journal of Experimental Algorithmics **13** (2008)
24. Alon, N., Halperin, D., Nechushtan, O., Sharir, M.: The complexity of the outer face in arrangements of random segments. In: Symposium on Computational Geometry 2008. (2008) 69–78

A Detailed Results of $\mathcal{D}/\mathcal{L}$ Ratio Experiments

This appendix contains all experiments concerning the ratio of the depth $\mathcal{D}$ of a DAG and the length $\mathcal{L}$ of the longest query path in the same DAG. In addition to those that are mentioned in Section 3.3 we also tested the special scenarios that we constructed in Section 3.1 in order to achieve lower bounds on the worst case ratio of $\mathcal{D}$ and $\mathcal{L}$. The set of segments is as depicted in Section 3.1. However, in the experiments here we choose random order of insertion.

In all experiments the two values hardly differ, that is, the largest ratio that we were able to observe was around 1.3. In the special scenarios, this value was even lower and in many cases $\mathcal{D}$ and $\mathcal{L}$ actually did not differ at all. However, for very large random scenarios, see Figure 5, $\mathcal{D}$ was always a bit larger than $\mathcal{L}$, but on the other hand the largest observed ratio even went down to less than 1.2.

This indicates that $\mathcal{D}$ and $\mathcal{L}$ behave sufficiently similarly. One can expect that an algorithm that rebuilds the DAG as soon as $\mathcal{L}$ becomes larger than $c \log n$ would actually rebuild more often than an algorithm that rebuilds as soon as $\mathcal{D}$ becomes larger than $1.3c \log n$, for some constant $c > 0$. This led us to venture Conjecture 1.

A.1 Experiments

We tested the ratio in the following scenarios:

1. Random line segments: Each segment was created from two random points in $[-1, 1]^2$. The number of generated segments was $\lfloor 1.5^k \rfloor$, for $6 \leq k \leq 19$. The reported results are the average of 20 builds of the search structure for the same random scenario. See Figure 5 (left).
2. Voronoi diagram of random points: For each scenario we took 2^k random sites for $6 \leq k \leq 15$. For each k we generated 10 different point sets and created the search structure 7 times, that is, the reported results are the average of 70 builds. See Figure 5 (right).
3. Lower bound construction for $O(\sqrt{n})$ ratio: We created the special scenarios according to the description in Section 3.1, each containing k^2 segments for $k \in \{10 \cdot 2^i | i \in \{1, \ldots, 6\}\}$. See Figure 6 (left)
4. Lower bound construction for $O(n/\log n)$ ratio: We created the special scenarios according to the description in Section 3.1, each containing 2^k segments for $8 \leq k \leq 17$. See Figure 6 (right)

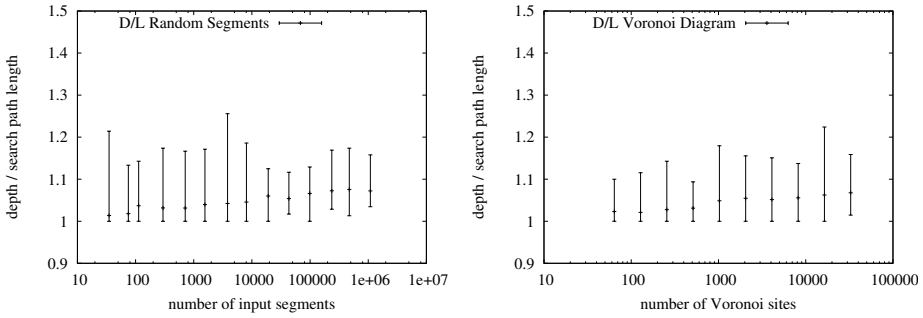


Fig. 5: $\mathcal{D}/\mathcal{L}$ for arrangement of random segments (left) and Voronoi Diagram of random sites (right). Plots show average value with error bars.

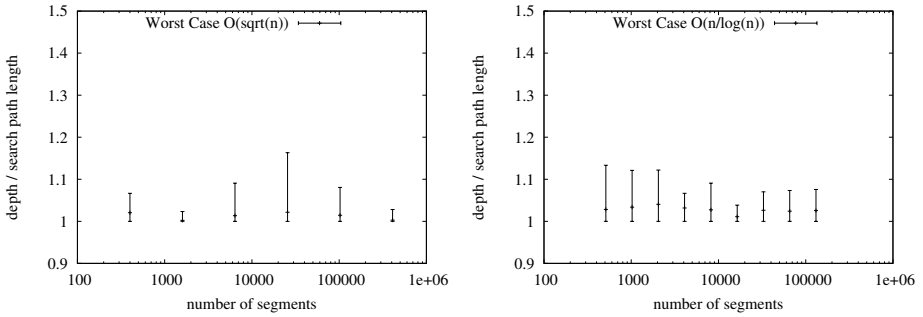


Fig. 6: $\mathcal{D}/\mathcal{L}$ for the example with worst case ratio $O(\sqrt{n})$ (left), and $O(n/\log n)$ (right). Plots show average value with error bars.

B Comparison to the CGAL's Landmarks Point Location

We emphasize that the new implementation of the trapezoidal-map random incremental construction for point location (RIC) performs better than all other point location methods available for CGAL arrangements.

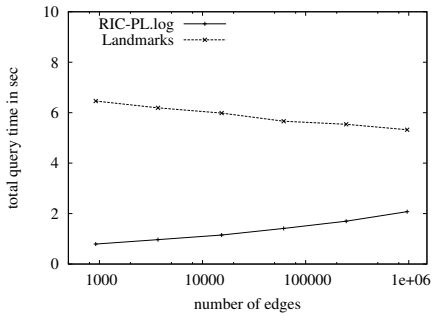


Fig. 7: Comparing the total query time for 50k queries in random subdivision of a varying size using both the CGAL Landmarks and the RIC point location methods.

Figure 7 displays the difference in the total query time in different arrangements of random segments using the RIC vs. the Landmarks (LM) point location. The landmarks generator in this experiment created landmarks on a $\lceil\sqrt{V}\rceil \times \lceil\sqrt{V}\rceil$ grid (V is the number of vertices in the arrangement). In [23] it is shown that for subdivisions of random segments the LM using the grid generator performs better than other point location methods implemented in CGAL, other than the RIC. As expected, the new RIC implementation outperforms the LM. Obviously, the RIC query time is logarithmic. The slight improvement of the query time of the LM can be explained by the fact that, at some point, while the number of input segments increases the average complexity of a face decreases, an effect that was for instance studied in [1].

References

1. Alon, N., Halperin, D., Nechushtan, O., Sharir, M.: The complexity of the outer face in arrangements of random segments. In: Symposium on Computational Geometry 2008. (2008) 69–78

C Computing the Depth of $\mathcal{A}(T)$

We would like to describe a linear space algorithm with $O(n \log n)$ runtime for computing the depth of a collection of open trapezoids with the following properties: their bases are y-axis parallel (vertical walls) and if the top or bottom curves of two different trapezoids intersect then the two curves overlap completely in their joined x-range. The depth of such a collection is the maximum number of trapezoids containing a common point, that is, we are only interested in points located on faces of the arrangement of all trapezoids. In Subsection C.1 we restate the algorithm of [17] such that the general position assumption can be dropped. The restated algorithm can handle rectangles with independently open or closed boundaries, and is more general than what we essentially need. Subsection C.2 defines a reduction from the collection of open trapezoids T into a collection R of open axis-parallel rectangles such that the maximum depth in $\mathcal{A}(R)$ is the same as the maximum depth in $\mathcal{A}(T)$. Finally, in Subsection C.3 we describe a modification for the restated algorithm such that it can compute the depth of the arrangement of all trapezoids created during the construction of the DAG.

C.1 An Algorithm for Computing the Depth of a Collection of Axis-aligned Rectangles

The algorithm of Alt & Scharf [17] is an $O(n \log n)$ algorithm that computes the depth of a collection of axis-aligned rectangles in general position, using $O(n)$ space. We give here a minor modification which does not assume general position, i.e., rectangles may share boundaries. Moreover, it can consider each of the four boundaries of a rectangle as either open or closed.

Given a set of finite rectangles, the set of all x-coordinates of the vertical sides of the input rectangles is first sorted. Let $x_1, x_2, ..., x_m$, $m < 2n$ be the sorted set of x-coordinates. The ordered set of intervals $\mathcal{I}$, is defined as follows; For $i \in 1, 2, ..., m-1$, the $2(i-1)$th and $2(i-1)+1$th intervals in the set $\mathcal{I}$ are $[x_i, x_i]$ and (x_i, x_{i+1}), respectively. The last interval is $[x_m, x_m]$. A balanced binary tree T is then constructed, holding all intervals in $\mathcal{I}$ in its leaves, according to their order in $\mathcal{I}$. An internal node represents the union of the intervals of its two children, which is a continuous interval. In addition, each internal node v stores in a variable $v.x$ the x-value of the merge point between the intervals of its two children. Since we extended the algorithm to support both open or closed boundaries, internal nodes also maintain, a flag indicating whether the merge point is to the left or to the right of the x-value.

According to the algorithm in [17], a sweep is performed using a horizontal line from $y = \infty$ to $y = -\infty$. The sweep-line events occur when a rectangle starts or ends, i.e., when top or bottom boundary of a rectangle is reached. Since the rectangles are not in general position, several events may share the same y-coordinate. In such a case, the order of event processing in each y-coordinate is as follows:

1. Closing rectangle with open bottom boundary events
2. Opening rectangle with closed top boundary events

3. Closing rectangle with closed bottom boundary events
4. Opening rectangle with open top boundary events

The order of event processing within each of these four groups in a specific y-coordinate is not important.

The basic idea of the algorithm is that each sweep event updates the appropriate leaves of the tree T (update the relevant leaves, spanning the covered intervals, with the current event). Therefore, each leaf holds a counter c for the number of covering rectangles in the current position of the horizontal sweep line. In addition, each leaf maintains in a variable c_m the maximal number of covering rectangles for this leaf seen so far. Clearly, the maximal coverage of an interval is the maximal c_m of all leaves. The problem with this naïve approach is that one such update can already take $O(n)$ time. Therefore, the key idea of [17] is that when updating an event of a rectangle whose x-range is (a, b), one should follow only two paths; the path to a and the path to b. The nodes on the path should hold the information of how to update the interval spanned by their children. In the end of the update the union of intervals spanned by the updated nodes (internal nodes and only 2 leaves) is (a, b).

In order to hold the information in the internal nodes each internal node holds the following variables:

l The additive update of rectangles that were opened or closed and cover the interval spanned by the left child of v since the last traversal of that child
r The additive update of rectangles that were opened or closed and cover the interval spanned by the right child of v since the last traversal of that child
l_m A counter which is used to count the maximum of the additive update for the left child since the last traversal of that child
r_m A counter which is used to count the maximum of the additive update for the right child since the last traversal of that child

A leaf, on the other hand, holds two variables:

c The coverage of the associated interval during the sweep until the last traversal on the leaf
c_m The maximum coverage of the associated interval during the sweep until the last traversal on the leaf

In realation to these values we define the following functions:

$$t(v) = \begin{cases} u.l + t(u) & \text{if } v \text{ is the left child of } u \\ u.r + t(u) & \text{if } v \text{ is the left child of } u \\ 0 & \text{if } v \text{ is the root} \end{cases}$$

$$t_m(v) = \begin{cases} max(u.l_m, u.l + t_m(u)) & \text{if } v \text{ is the left child of } u \\ max(u.r_m, u.r + t_m(u)) & \text{if } v \text{ is the left child of } u \\ 0 & \text{if } v \text{ is the root} \end{cases}$$

At any point of the sweep the following two invariants hold for every leaf ℓ and its associated interval I:

- The current coverage of I is: $\ell.c + t(\ell)$
- The maximal coverage of I that was seen so far is: $\max(\ell.c_m, \ell.c + t_m(\ell)$

Updating the structure with an event is done as follows: Let I be the x-interval spanned by the processed rectangle creating the event. Depending on whether the rectangle starts or ends, we set a variable $d = 1$ or $d = -1$, respectively. We follow the two search paths to the leftmost leaf and the rightmost leaf that are covered by I. In the beginning the two paths are joined until they split, for every node w on this path (including the split node) we can ignore d and simply update the tuple $(w.l, w.r, w.l_m, w.r_m)$ using $t(w)$ and $t_m(w)$ according to the invariants stated above. Note that this process needs to clear the corresponding values in the parent node as otherwise the invariants would be violated.[6] After the split the paths are processed separately, we discuss here the left path, the behavior for the right path is symmetric. Let v be a node on the left path. As long as v is not a leaf we update $(v.l, v.r, v.l_m, v.r_m)$ as usual. However, if the path continues to the left we also have to incorporate d into $v.r$ and $v.r_m$ as the subtree to the right is covered by I. If v is a leaf we simply update $v.c$ and $v.c_m$ using $t(v), t_m(v)$ and d. A more detailed description (including pseudo code) can be found in [17]. In total, this process takes $O(\log n)$ time.

Finally, in order to find the maximal number of rectangles covering an interval a final propagation from root to leaves is needed, such that all l, r, l_m, r_m values of internal nodes are cleared. This is done using one traversal on T. Now, the maximal number of rectangles covering an interval is the maximal c_m of all leaves of T.

Clearly, the running time of the algorithm is $O(n \log n)$, since constructing the tree and sorting the y-events takes $O(n \log n)$ time. Updating each of the $2n$ y-events takes $O(\log n)$ time, and the final propagation of values to the leaves takes $O(n)$ time. The algorithm uses $O(n)$ space.

We remark that the above algorithm is not optimal in memory usage. A more efficient variant which stores less variables in the nodes of the tree can be easily implemented.

C.2 A Depth Preserving Reduction

Let T be a collection of open trapezoids with y-axis parallel bases with the following property: if the top or bottom curves of two different trapezoids intersect then the two curves overlap completely in their joined x-range. Let $\mathcal{A}(T)$ denote the arrangement of the trapezoids in T. We describe a reduction from T to R, where R is a collection of axis-parallel rectangles, such that the maximum depth in $\mathcal{A}(R)$ equals to the maximum depth in $\mathcal{A}(T)$.

[6] Please note that using $t(w)$ and $t_m(w)$ here takes constant time since we only need to access the parent node as all previous nodes on the path towards the root are already processed.

The following observation is by [16]:

Observation 2. *Let S be a set of interior disjoint x-monotone curves. There exists a partial order on S, such that if the curves are moved one at a time to the direction of $y = -\infty$ according to this order, then each curve can be moved without (interior) intersecting any of the remaining curves. This order can be extended to a total order.*

Definition 2. *Let C be a set of interior disjoint x-monotone curves. For two such curves $cv_i, cv_j \in C$, let the open interval (a, b) be the x-range of cv_i and the open interval (c, d) be the x-range of cv_j. We define the total order $\prec$ as follows:*
If $x\text{-range}(cv_i) \bigcap x\text{-range}(cv_j) = \emptyset$ then:
$cv_i \prec cv_j \Leftrightarrow b \leq c$
If $x\text{-range}(cv_i) \bigcap x\text{-range}(cv_j) \neq \emptyset$ then:
$cv_i \prec cv_j \Leftrightarrow cv_i(x) < cv_j(x)$ *for some* $x \in x\text{-range}(cv_i) \bigcap x\text{-range}(cv_j)$.

Definition 3. *Let Ord denote a function $Ord : C \to \{1, ..., n\}$ returning the order of a given x-monotone curve $cv \in C$ when sorting C according to $\prec$.*

Definition 4. *We define a reduction from T to R as follows; Every trapezoid $t \in T$ is reduced to a rectangle $r \in R$, such that:*

- *t and r have the same x-range,*
  i.e. $(left(t) = left(r))$ and $(right(t) = right(r))$, where $left$ and $right$ denote the left x-value and the right x-value of t (or r), respectively.
- *The top and bottom edges of r (accessible by top and bottom methods) lie on $y = Ord(top(t))$ and $y = Ord(bottom(t))$, respectively.*

As shown in [13], one can partition the plane into vertical slabs by passing a vertical line through every endpoint of the subdivision, and then partition each slab into regions by intersecting it with all possible curves in the subdivision. This defines a decomposition of the plane into at most $2(n + 1)^2$ regions.

Lemma 2. *Let $Regions(arr)$ denote the collection of regions of an arrangement arr, as defined above. For any region $a_t \in Regions(\mathcal{A}(T))$ let $a_r \in Regions(\mathcal{A}(R))$ be the matching rectangular region to a_t. The collection $Regions(\mathcal{A}(R))$ of all such rectangular regions spans the plane.*

Proof. Trivial. The slabs remain the same and within each slab the rectangular regions remain adjacent. □

Lemma 3. *Let $a_t \in Regions(\mathcal{A}(T))$ be a region, whose matching region is $a_r \in Regions(\mathcal{A}(R))$. The number of rectangles in R that cover a_r is at least the number of trapezoids in T that cover a_t. In other words, for every $t \in T$ that covers a_t its matching rectangle $r \in R$ covers a_r.*

Proof. Let $\{t_1, t_2, ..., t_m\} \subseteq T$ be the set of trapezoids, ordered by creation time, such that for every $i \in \{1, ..., m\}$, t_i covers a_t. Let $\{r_1, r_2, ..., r_m\} \subseteq R$ be the set

of matching rectangles, such that r_i matches t_i for $i \in \{1...m\}$. For any t_i, since t_i covers a_t we get that x-range$(a_t) \subseteq x$-range(t_i). By Definition 4 the x-ranges remain the same after the reduction, and therefore x-range$(a_r) \subseteq x$-range(r_i). Since t_i covers a_t then we also get that in the shared x-range $top(t_i)$ is above or on $top(a_t)$ and $bottom(t_i)$ is below or on $bottom(a_t)$. According to Definition 4, it immediately follows that $Ord(top(t_i)) \geq Ord(top(a_t))$. In other words, $top(r_i)$ is above or on $top(a_r)$. Similarly, $bottom(r_i)$ is below or on $bottom(a_r)$. We conclude that r_i covers a_r. □

Lemma 4. *Let $a_r \in Regions(\mathcal{A}(R))$ be a rectangular region, whose matching region is $a_t \in Regions(\mathcal{A}(T))$. The number of trapezoids in T that cover a_t is at least the number of rectangles in R that cover a_r. In other words, for every $r \in R$ that covers a_r its matching trapezoid $t \in T$ covers a_t.*

Proof. Let $\{r_1, r_2, ..., r_m\} \subseteq R$ be the set of rectangles, such that for every $i \in \{1, ..., m\}$, r_i covers a_r. Let $\{t_1, t_2, ..., t_m\} \subseteq T$ be the set of matching trapezoids, such that t_i matches r_i for $i \in \{1...m\}$. Proving that for any $i \in \{1...m\}$, t_i covers a_t, is done symmetrically to the proof of Lemma 3. □

Combining Lemma 3 and Lemma 4 we conclude that the number of trapezoids in T that cover a region a_t equals to the number of rectangles in R that cover a_r, which is the matching region to a_t. The covering rectangles are the reduced trapezoids in the set of trapezoids covering a_t. Since both $Regions(\mathcal{A}(T))$ and $Regions(\mathcal{A}(R))$ span the plane (Lemma 2), we get the following theorem.

Theorem 4. *Let T be a collection of open trapezoids with the following properties: their bases are y-axis parallel (vertical walls) and if the top or bottom curves of two different trapezoids intersect then the two curves overlap completely in their joined x-range. Let $\mathcal{A}(T)$ denote the arrangement of the trapezoids in T. T can be reduced to a collection of open axis-parallel rectangles R, such that the maximum depth in $\mathcal{A}(R)$ equals to the maximum depth in $\mathcal{A}(T)$.*

C.3 Modification of Alt & Scharf

Based on the correctness of the reduction described in Subsection C.2 we can extended the basic algorithm presented in Subsection C.1 to support not only collections of axis-aligned rectangles but also collections of open trapezoids with y-axis parallel bases (vertical walls) and non-intersecting top and bottom boundaries (if they intersect then they overlap completely in their joined x-range). The only part of the basic algorithm that should change is the top-to-bottom sweep. Therefore, the simple predicate in [17] that is used for sorting the y-events should be replaced with a new predicate that compares according to the reverse order of $\prec$, as given in Definition 2.

Please note that for simplicity we assumed that no two distinct endpoints in the original subdivision have the same x-value. However, if this is not the case, a lexicographical compare can be used on the endpoints of the curves in order to define the order of the induced vertical walls.